3. FORMULA : Composing and Transforming

Ethan Jackson, Nikolaj Bjørner and Wolfram Schulte
Research in Software Engineering (RiSE), Microsoft Research

Dirk Seifert, Markus Dahlweid and Thomas Santen
European Microsoft Innovation Center (EMIC), Microsoft Research

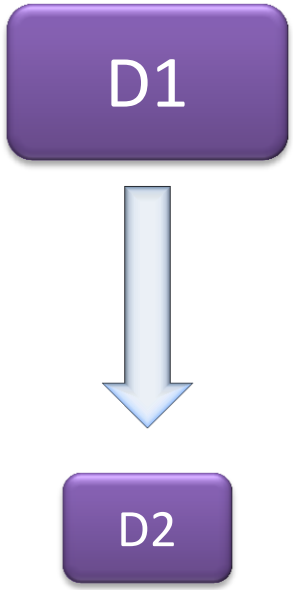
i. Composing Domains

http://research.microsoft.com/formula

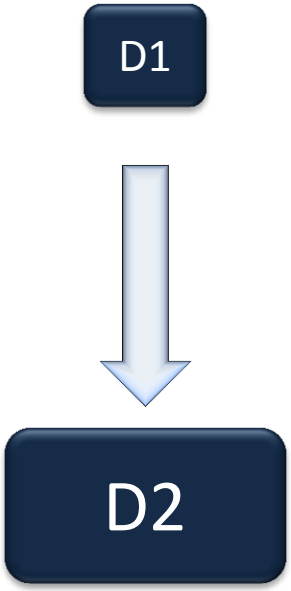
Composition

Composition allows us to modularly defined and build abstractions.

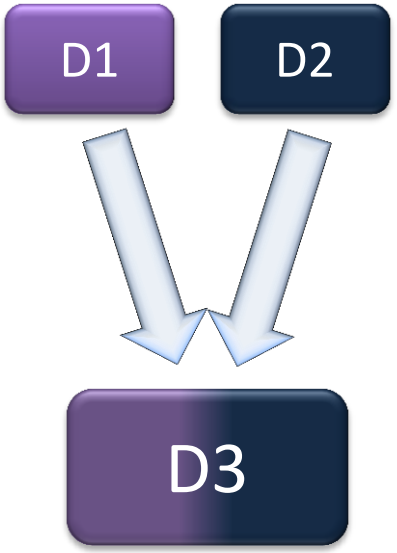
FORMULA supports several styles of formal composition.



Abstractions with more constraints (fewer models).



Abstractions with more primitives (more models).



Abstractions that combine the ADTs and constraints of others

Extends

Safe composition and importation of domains

domain D extends $D_1, \ldots, D_n\{\ \}$.

D contains the union of all data type declarations of extended domains.

$$decl(D) \supseteq \bigcup_i decl(D_i)$$

And the union of logic programs of extended domains.

$$prog(D) \supseteq \bigcup_i prog(D_i)$$

The conforms query is a conjunction of queries

$$conforms_D \stackrel{\text{def}}{=} local \wedge \bigwedge_i conforms_{D_i}$$

Extends

An `extends` composition is only legal if all conflicting data type declarations are semantically equivalent:

$$\forall\, \tau ::= decl_1, \tau ::= decl_2.\ [\![decl_1]\!] = [\![decl_2]\!]$$

And the semantics of conforms is preserved:

$$\forall M \subset Any_D.$$
$$conforms_{D_i} \in knows_D(M) \Rightarrow$$
$$conforms_{D_i} \in knows_{D_i}(M \cap [\![Any_{D_i}]\!])$$

These conditions are checked by the compiler; the second is a conservative approximation.

Extends

Certain patterns of extends guarantee relationships between domains:

```
domain D extends D1
{
    //// More constraints,
    //// but no new primitives
}
```

$$models(D) \subseteq models\ (D_1)$$

```
domain D extends D1
{
    //// New primitives,
    //// but no new constraints on D1 primitives
}
```

$$models(D) \supseteq models\ (D_1)$$

```
domain D extends D1, D2
{
    //// Nothing inside D;
    //// D1 and D2 have disjoint constructors
}
```

$$models(D) \approx models\ (D_1) \times models(D_2)$$

Examples – Graph Classes

Here are some graph classes using `extends`.

```
domain DAG extends Digraph
{
    conforms := fail path(x, x).
}
```

Constraint from Digraph implicitly here

```
domain ColoredGraphs extends Digraph
{
    primitive ColorMap ::= (v: V, c: Integer).
}
```

Would typically add a Function annotation here.

Examples – Graph Classes

This composition yields DAGs with colored vertices.

```
domain ColoredDAG extends Digraph, ColoredGraphs
{  }
```

Not equivalent to a product, because non-disjoint function symbols.

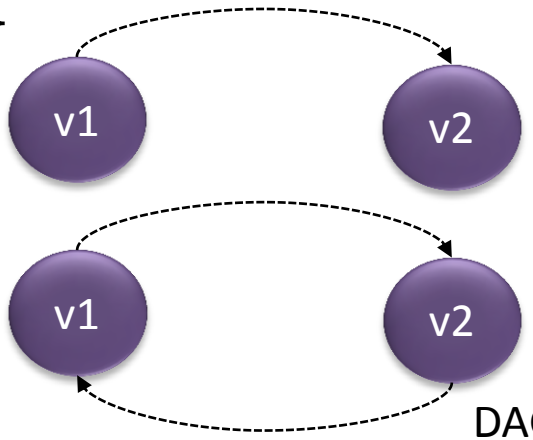
Examples – Graph Classes

This extension violates the use of extends and is rejected.

```
domain BadClass extends DAG
{
    path(x, y) :- x is V, y is V.
}
```

Rejected because creating new paths effects DAG.conforms

```
model M of BadClass
{
    v1 is V(1) v2 is V(2)
    E(v1, v2)
}
```



DAG.conforms = false

```
model M of DAG
{
    v1 is V(1) v2 is V(2)
    E(v1, v2)
}
```

DAG.conforms = true

Includes - An Escape Hatch

Lexical inclusion, with same check on conflicting type declarations.
No other checks and no strong guarantees on relationships between domains.

```
domain DAGorLoop includes DAG
{
    loop := E(x, x).
    conforms := DAG.conforms | loop.
}
```

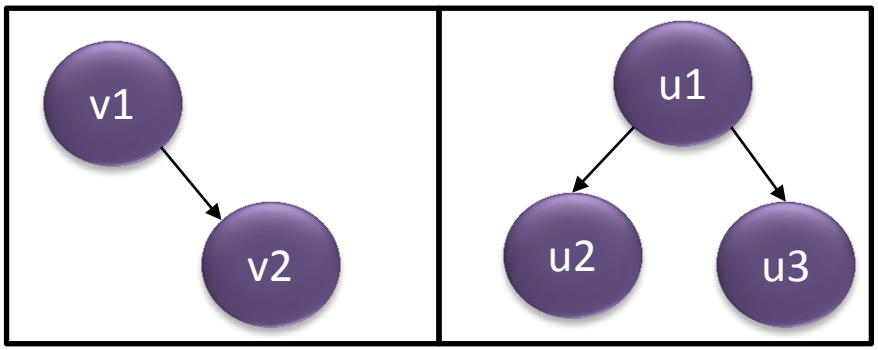
Define conformance however we like…

Renaming with “As”

What if we want models that are pairs of DAGs?

```
domain DAGPair
{
    primitive V1 ::= (id: Integer).
    primitive E1 ::= (src: V1, dst: V1).
    primitive V2 ::= (id: Integer).
    primitive E2 ::= (src: V2, dst: V2).
    path1 ::= (beg: V1, end: V1).
    path2 ::= (beg: V2, end: V2).
    path1(x, z) :- E1(x, z); path1(x, y), path1(y, z).
    path2(x, z) :- E2(x, z); path2(x, y), path2(y, z).
    //// More stuff....
}
```

◀ Two copies of the DAG specification, but with different names



Renaming with “As”

The renaming operator prepends all user types and query names, and then applies this to all rule and query declarations.

```
domain DAGPair2 extends DAG as G1, DAG as G2 { }
```

Vertices and edges from the first graph are `G1.V` and `G1.E`. The second graph are `G2.V` and `G2.E`.

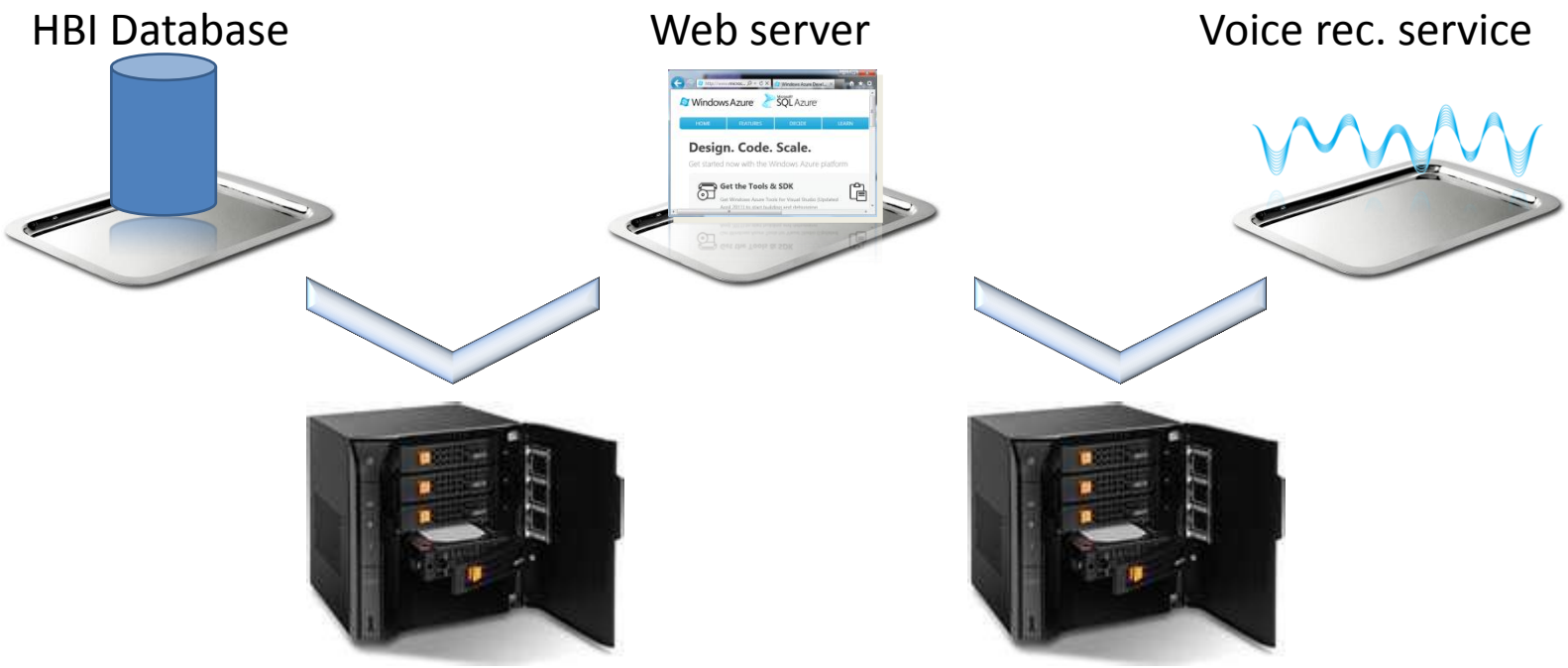
$$models(DAGPair2) \approx models\ (DAG) \times models(DAG)$$

ii. Several Examples

http://research.microsoft.com/formula

Example 1 – Data Center Configuration

How can a set of services be feasibly deployed in a data center?
(Functional correctness of services doesn't help)



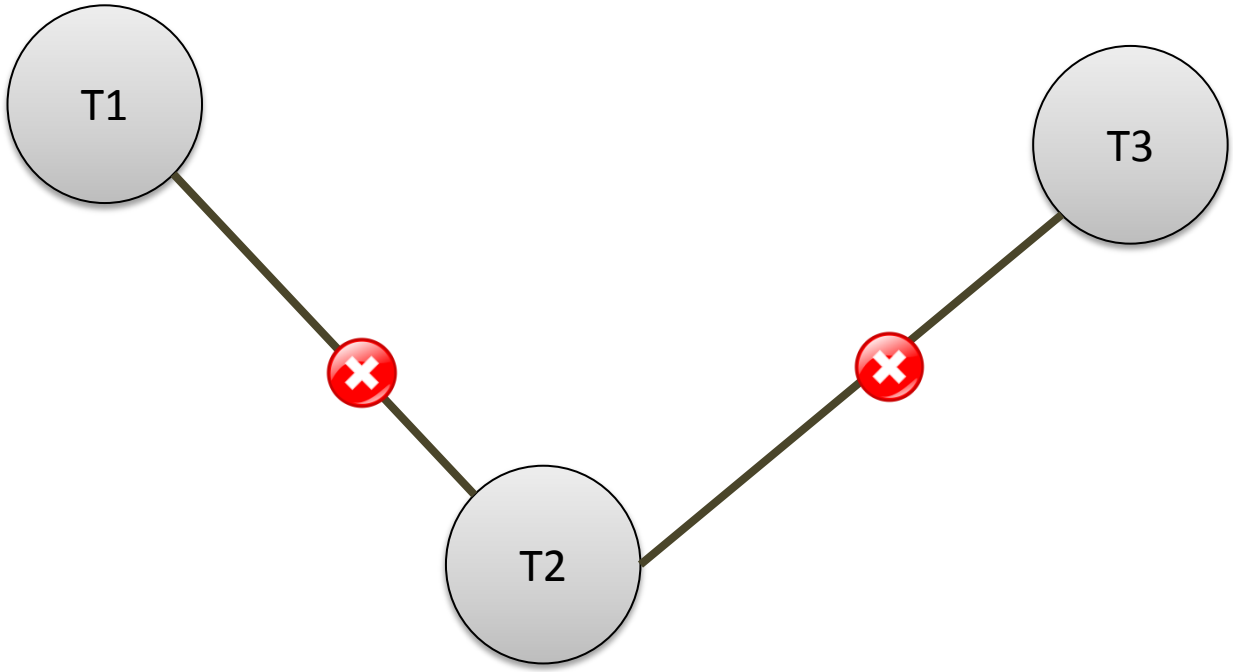
Can't do this; literally illegal in some cases.

Can't do this; may overload CPU.

Need some model of conflicts between services, but not necessarily implementation details.

Applications (I)

For simplicity, assume an application is just a task.
Two tasks can be in conflict, meaning they should not execute on the same node.



Applications (II)

The "domain" keyword starts and abstraction

Data type constructors with labeled arguments and type constraints

```
domain Applications
{
        App ::= (id: String).
        [Closed]
        Conflict ::= (t1: App, t2: App).
}
```

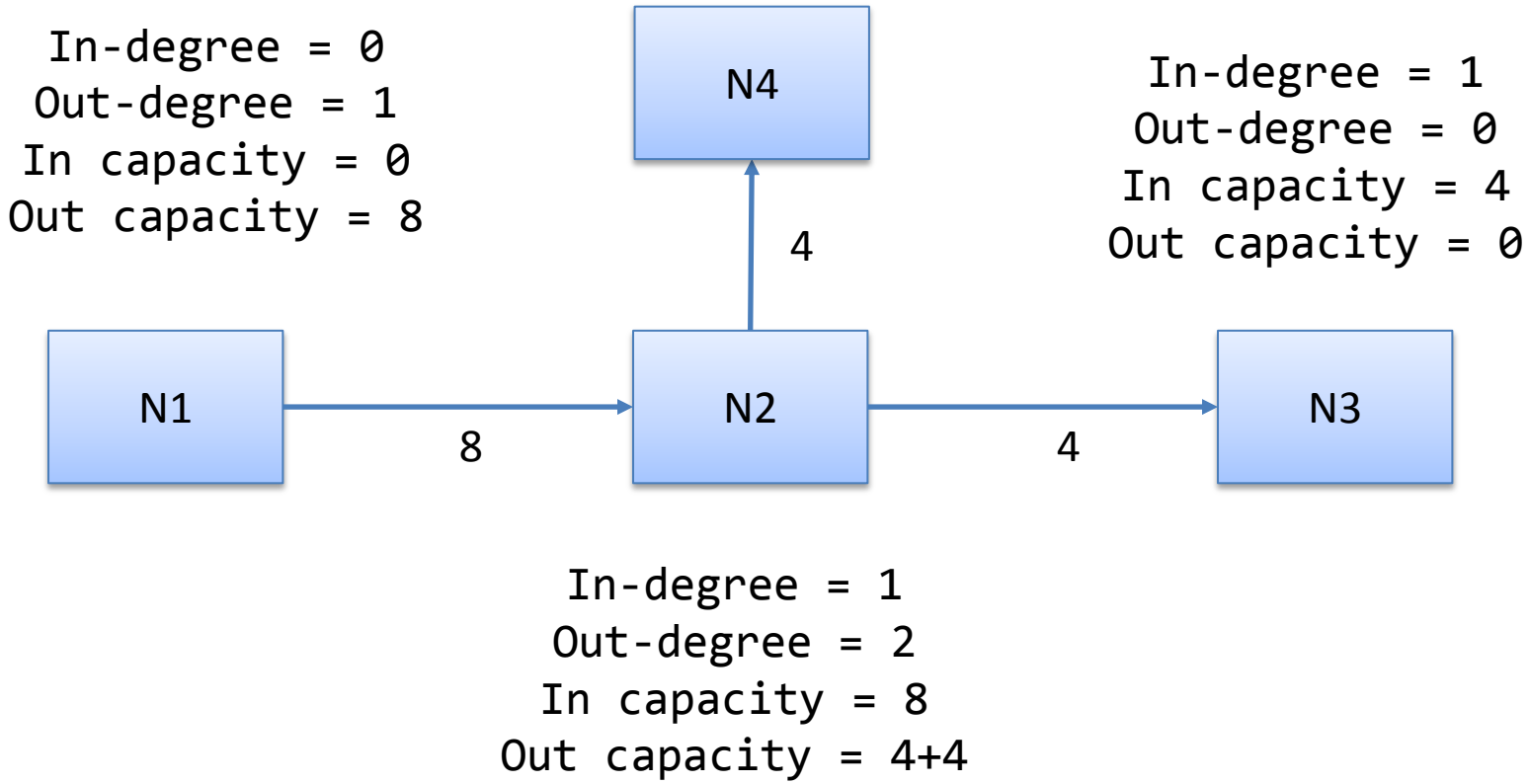
A "model" is claim of conformance

And a set terms built using data type constructors

```
model ApplicationModel of Applications
{
        t1 is App("HBI Database")
        t2 is App("Web Server")
        t3 is App("Voice Recognition")
        Conflict(t1, t2)
        Conflict(t2, t3)
}
```

The Cloud (I)

Nodes are connected by channels with communication capacities.
No node can support more than two incoming and outgoing channels.
Capacities must be balanced on node with incoming and outgoing channels.



The Cloud (II)

Special annotations for common constraints

Named "queries" can be treated like Boolean variables.

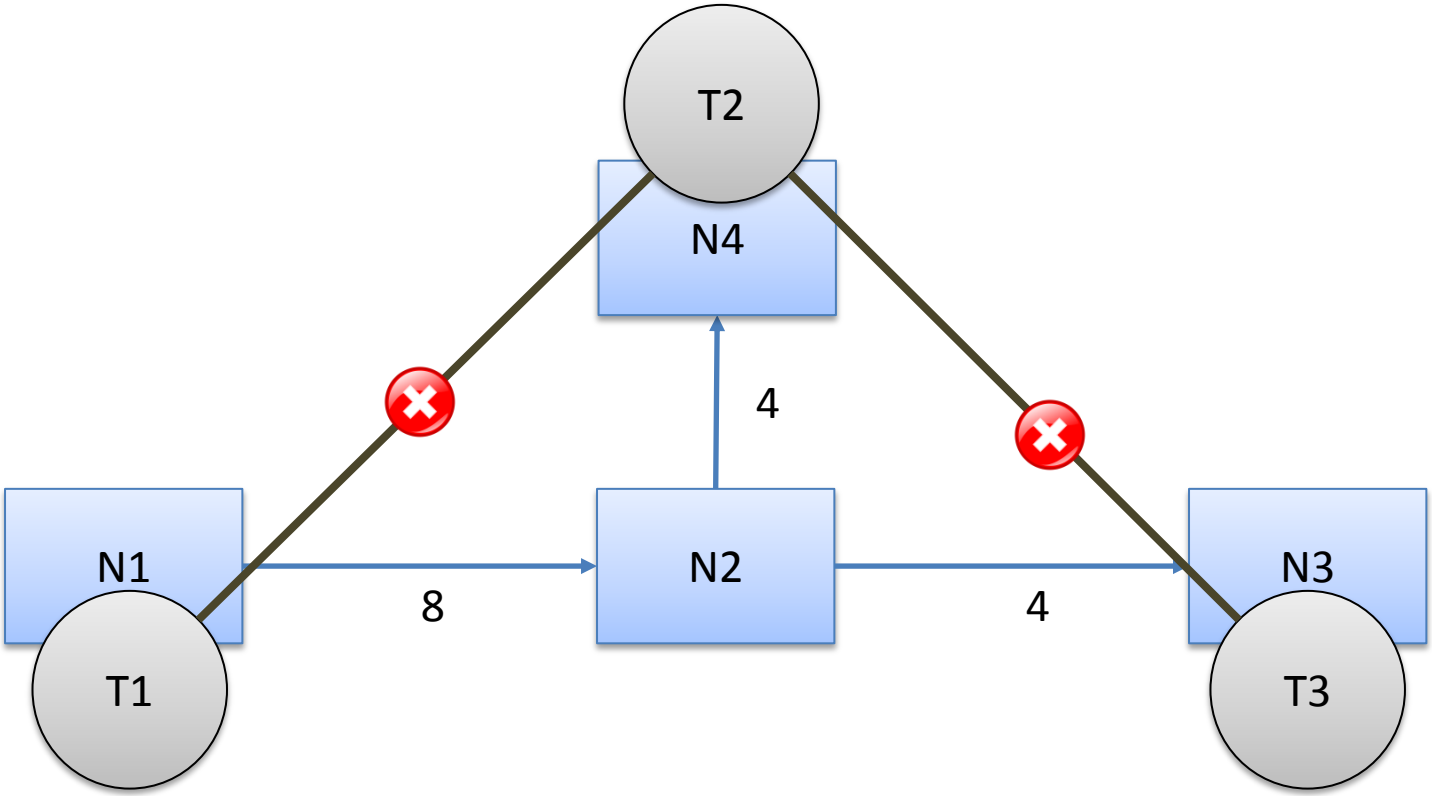
Rules derive complex information

The "conforms" query determines the models.

```
domain Cloud {
  Node    ::= (id: Integer).
  [Closed(fromNode, toNode)]
  [Unique(fromNode, toNode -> cap)]
  Channel ::= (fromNode: Node, toNode: Node,
               cap: PosInteger).

  bigFanIn  := n is Node, count(Channel(_,n,_)) > 2.
  bigFanOut := n is Node, count(Channel(n,_,_)) > 2.

  mustBal(n) :- Channel(_,n,_), Channel(n,_,_).

  clog := mustBal(n),
          sum(Channel(_,n,_),2) !=
          sum(Channel(n,_,_),2).

  conforms := !(bigFanIn | bigFanOut | clog).
}
```

Deployments (I)

Tasks should be place on nodes so all conflict constraints are respected.



Deployments (II)

The "extends" keyword safely composes

Only need to write the new constraints.

```
domain Deployment extends Applications, Cloud
{
  [Closed] [Function(fromApp -> toNode)]
  Binding ::= (fromApp: App, toNode: Node).

  inConflict := Binding(t1, n), Binding(t2, n),
                Conflict(t1, t2).

  conforms   := !inConflict.
}
```

Simplified example, but not an easy one:

A coloring problem,
A forbidden-subgraph problem
Linear arithmetic problems

Realistic problems contain constraints like these.

Solve in Any Direction

The user constructs a partial model to represent the degrees of freedom in the problem. Degrees of freedom can be anywhere.

Entities that must be in any solution

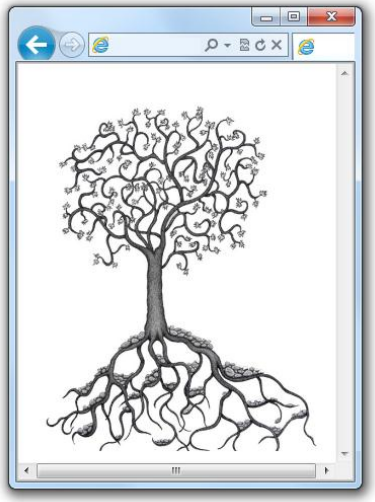
Explicit degrees of freedom. There are also implicit degrees of freedom, like binding.

```
partial model Ex of Deployment
{
  t1 is App("HBI Database")
  t2 is App("Web Server")
  t3 is App("Voice Recognition")
  Conflict(t1, t2)
  Conflict(t2, t3)

  n1 is Node(1)
  n2 is Node(2)
  n3 is Node(3)
  c1 is Channel c2 is Channel c3 is Channel
  c4 is Channel c5 is Channel c6 is Channel
  c7 is Channel c8 is Channel c9 is Channel
}
```

Example 2 – Web-Service Integration

When (correct) web-services are mashed-up can new vulnerabilities arise?

Genealogy site



Policy for computing genealogical relationships

Genealogy data to make friend recommendations

Can Evil Eve get recommended as friend to Bob?

Social networking site

Policy for computing friend recommendations

Full static verification would be very difficult.
Abstracting to the data policies focuses the problem.

Rules of the Social Network

```
domain Principals
{
   Person ::= (first: String, last: String).
}

domain SocNetwork extends Principals
{
  [Closed]
  Friend         ::= (Person, Person).
  isDirectFriend ::= (Person, Person).
  isFofF         ::= (Person, Person).

  isDirectFriend(p1, p2),
  isDirectFriend(p2, p1) :- Friend(p1, p2).
  isFofF(p1, p2)         :- isDirectFriend(p1, p2).
  isFofF(p1, p2)         :- isFofF(p1, p), isFofF(p, p2).
  recFriend(p1, p2)      :- isFofF(p1, p2), p1 != p2,
                            fail isDirectFriend(p1, p2).

  recAndNotFofF := recFriend(p1, p2), p1 != p2, fail isFofF(p1, p2).
}
```

Can Eve Do Anything Suspicious?

Maybe she can even make up her last name…

```
partial model Net of SocNetwork
{
   pEve   is Person("Eve", _)
   pBob   is Person("Bob", "Bob")
   pChuck is Person("Chuck", "Chuck")
   pAlice is Person("Alice", "Alice")
   Friend(pAlice, pBob)
   Friend(pBob, pChuck)
}
```

The Family Tree Website

```
domain FamilyTree extends Principals
{
   [Closed]
   Database  ::= (Person).
   isRelated ::= (Person, Person).

   isRelated(p1, p2) :- Database(p1), Database(p2),
                        p1.last = p2.last, p1 != p2.
}
```

The Integration

```
domain Integration extends SocNetwork, FamilyTree
{
   recFriend(p1, p2) :- isRelated(p1, p2).
}


[Introduce(Database, 4)]
partial model NetInt of Integration
{
   pEve   is Person("Eve", _)
   pBob   is Person("Bob", "Bob")
   pChuck is Person("Chuck", "Chuck")
   pAlice is Person("Alice", "Alice")
   Friend(pAlice, pBob)
   Friend(pBob, pChuck)
}
```

A Suspicious Scenario

Not a bug in the usual sense, but a scenario that should be carefully considered.

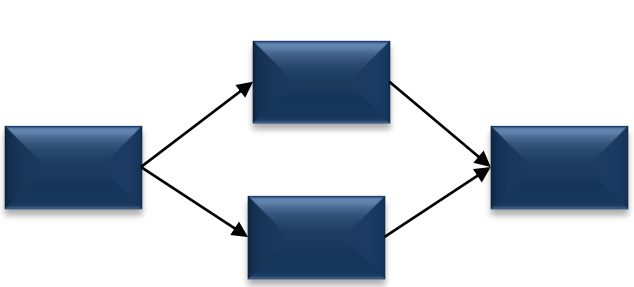
```
model NetInt_1 of Integration at "../WICSAExamples.4ml"
{
   Person("Alice","Alice")
   Person("Bob","Bob")
   Person("Chuck","Chuck")
   Person("Eve","Chuck")

   Friend(Person("Alice","Alice"),Person("Bob","Bob"))
   Friend(Person("Bob","Bob"),Person("Chuck","Chuck"))

   Database(Person("Bob","Bob"))
   Database(Person("Chuck","Chuck"))
   Database(Person("Eve","Chuck"))
}
```

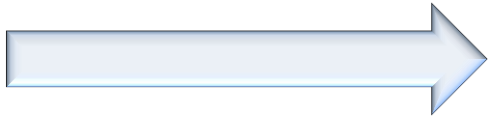
iii. Transformations

http://research.microsoft.com/formula

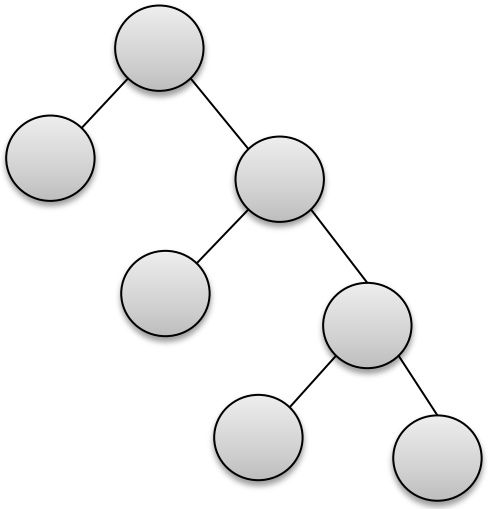
Why Transformations?

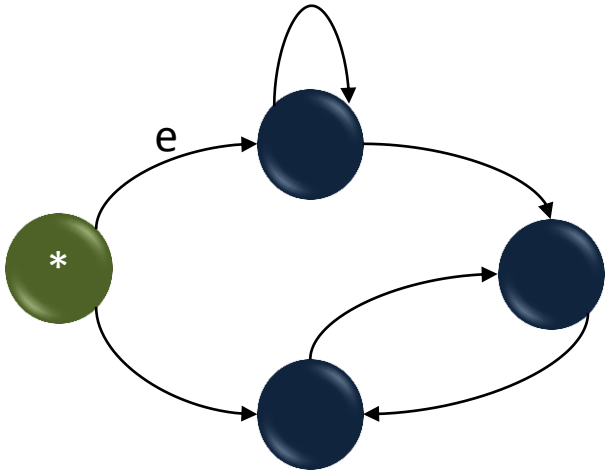
Often needed to change levels of abstraction, or to define operational steps.

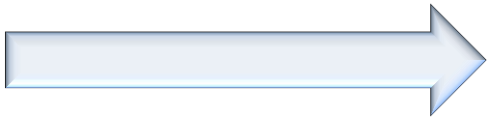




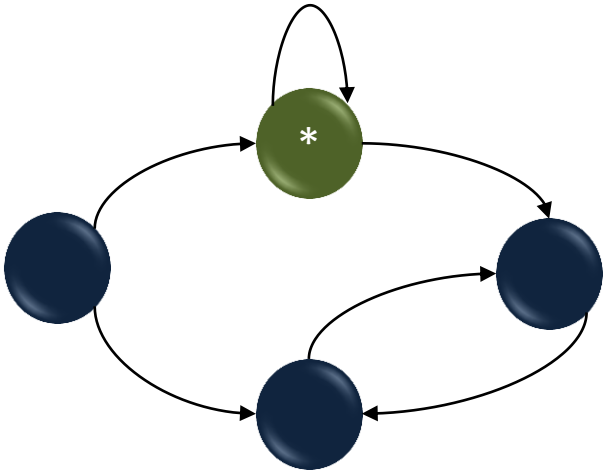
Compiler from dataflow systems to expression trees







Evolve the modeled state in response to external stimulus.



Transformations in FORMULA

Transformations are defined using the same language core.

A transformation from n models to m models has a signature:

transform T from D_1 as in_1 … D_n as in_n
to D'_1 as out_1 … D'_m as out_m { }

The transformation contains the renamed (products) version of the input and output domains.

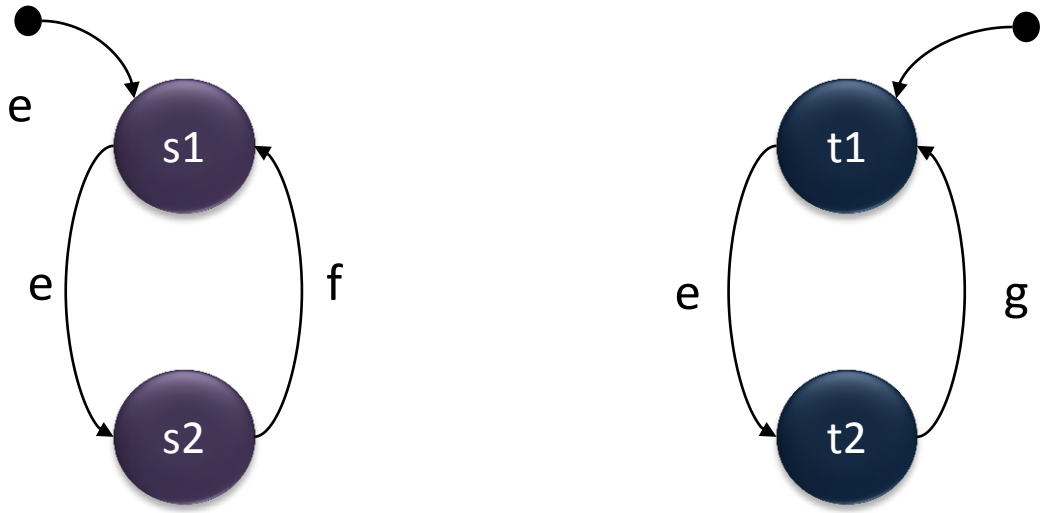
The input models close the logic program, the knowledge is computed, and the output models are projected from the knowledge.

Finite Automata

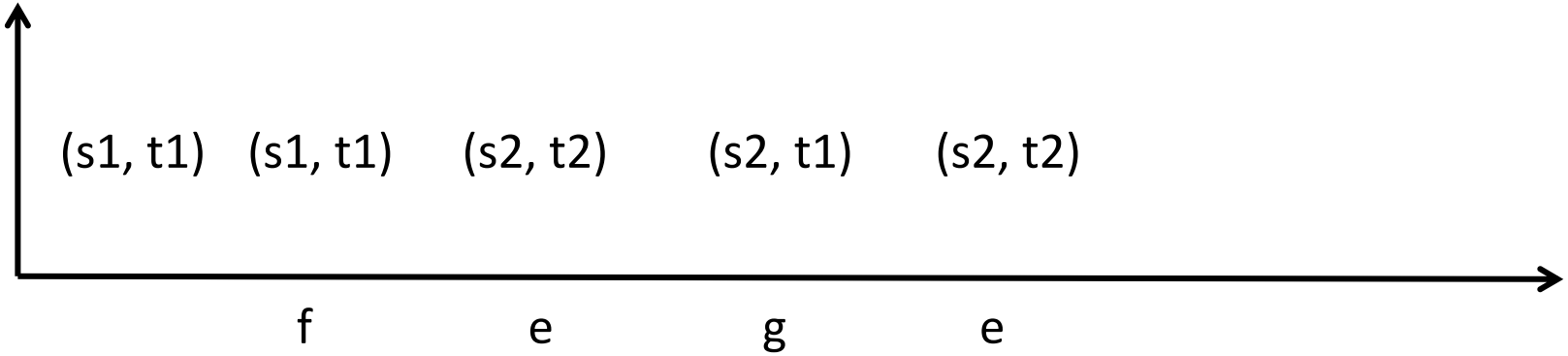
```
domain FA
{
    primitive State ::= (name: String).
    primitive Event ::= (name: String).
    [Closed(cur, trig, nxt)]
    primitive Transition ::= (cur: State, trig: Event,
                              nxt: State).
    [Closed(st)]
    primitive Current ::= (st: State).
    conforms := c is Current.
}

domain DFA extends FA
{
    tooManyCurrents := c1 is Current, c2 is Current, c1 != c2.
    nonDeterTrans   := t1 is Transition, t2 is Transition,
                       t1.cur = t2.cur,
                       t1.trig = t2.trig,
                       t1.nxt != t2.nxt.
    conforms := !(tooManyCurrents | nonDeterTrans).
}
```

Finite Automata

What does it mean to run these two automata in parallel?

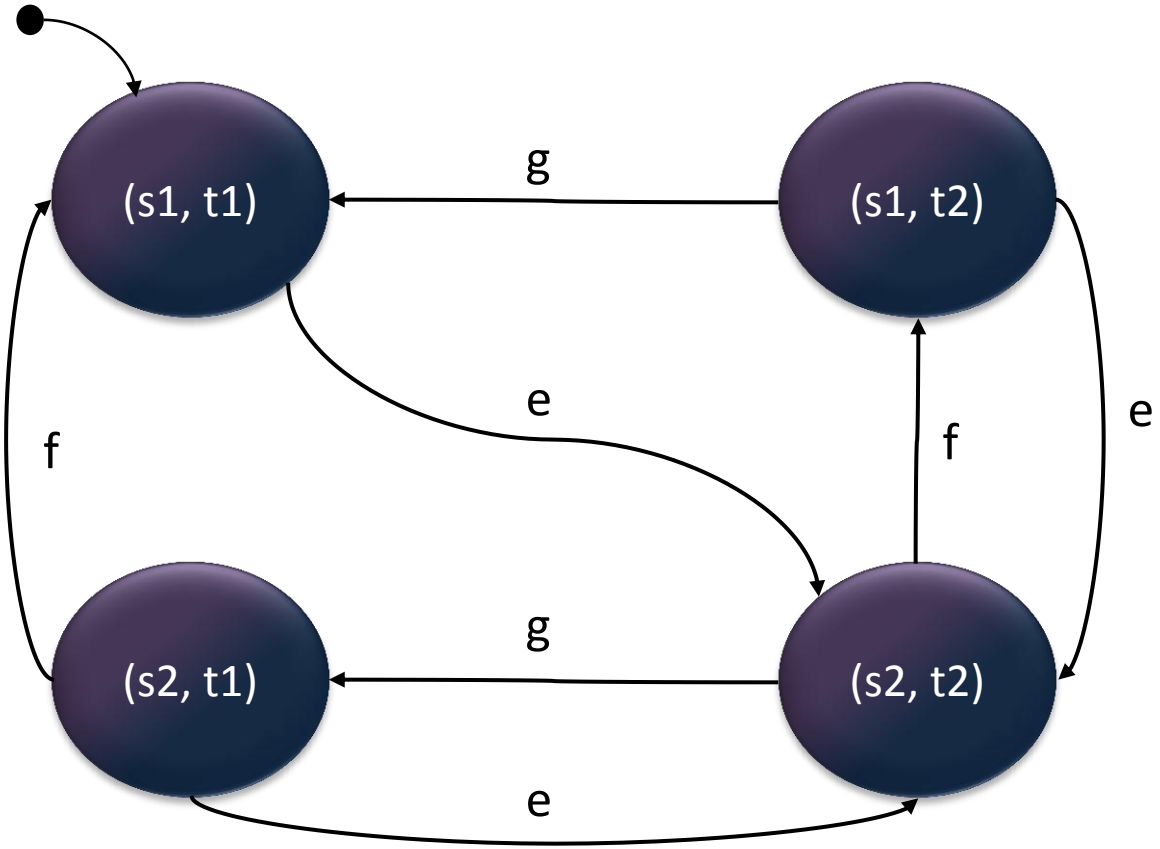


One interpretation is that they synchronize on common events, but run asynchronously otherwise.



Product Automata

This synchronization behavior can be described by a product automaton



Product Automata

The synchronous product constructs the product automaton

$$\mathcal{A} \otimes \mathcal{A}' \stackrel{def}{=} (I_\otimes, Q_\otimes, \Sigma_\otimes, \rightarrow_\otimes).$$

where

1. Initial states: $I_\otimes \stackrel{def}{=} I_\mathcal{A} \times I_{\mathcal{A}'}$,
2. States: $Q_\otimes \stackrel{def}{=} Q_\mathcal{A} \times Q_{\mathcal{A}'}$,
3. Events: $\Sigma_\otimes \stackrel{def}{=} \Sigma_\mathcal{A} \cup \Sigma_{\mathcal{A}'}$,
4. Transitions:

$$\begin{aligned} \rightarrow_\otimes \stackrel{def}{=} & \{(s,t) \stackrel{\alpha}{\rightarrow}_\otimes (s',t) \mid s \stackrel{\alpha}{\rightarrow}_\mathcal{A} s' \wedge t \stackrel{\alpha}{\not\rightarrow}_{\mathcal{A}'}\} \cup \\ & \{(s,t) \stackrel{\alpha}{\rightarrow}_\otimes (s,t') \mid t \stackrel{\alpha}{\rightarrow}_{\mathcal{A}'} t' \wedge s \stackrel{\alpha}{\not\rightarrow}_\mathcal{A}\} \cup \\ & \{(s,t) \stackrel{\alpha}{\rightarrow}_\otimes (s',t') \mid s \stackrel{\alpha}{\rightarrow}_\mathcal{A} s' \wedge t \stackrel{\alpha}{\rightarrow}_{\mathcal{A}'} t'\}. \end{aligned}$$

Product Automata

Define a new domain for representing product automata.

```
domain ProdFA
{
    primitive State ::= (nameLeft: String, nameRight: String).
    primitive Event ::= (name: String).
    [Closed(cur, trig, nxt)]
    primitive Transition ::= (cur: State, trig: Event,
                              nxt: State).
    [Closed(st)]
    primitive Current ::= (st: State).
    conforms := c is Current.
}
```

Product Automata

Define a transformation from two automata to a product automata.

```
transform SyncProd from DFA as in1, DFA as in2 to ProdFA as out1
{
    out1.Event(x) :- in1.Event(x); in2.Event(x).
    out1.State(s, t) :- in1.State(s), in2.State(t).
    out1.Current(State(s.name, t.name)) :-
        in1.Current(s), in2.Current(t).

    out1.Transition(State(s, t), Event(a), State(sp, t)) :-
        in1.Transition(State(s), Event(a), State(sp)),
        in2.State(t), fail in2.Transition(State(t), Event(a), _).
    out1.Transition(State(s, t), Event(a), State(s, tp)) :-
        in2.Transition(State(t), Event(a), State(tp)),
        in1.State(s), fail in1.Transition(State(s), Event(a), _).
    out1.Transition(State(s, t), Event(a), State(sp, tp)) :-
        in1.Transition(State(s), Event(a), State(sp)),
        in2.Transition(State(t), Event(a), State(tp)).
}
```

Proving Properties

Model finding can be used to prove properties about transformations.

```
transform SyncProd from DFA as in1, DFA as in2 to ProdFA as out1
{
   //// Previous code
   NFAOut := out1.Transition(s, e, sp), out1.Transition(s, e, spp),
             sp != spp.
   notDFAPreserving := in1.DFA.conforms & in2.DFA.conforms &
             out1.ProdFA.conforms & NFAOut.
}
```

Can show that synchronous product **preserves** determinism by absence of counter examples up to some size.

Proving Properties

On the other hand, the shuffle (asynchronous) product, does not preserve determinism.

```
transform ShuffleProd from DFA as in1, DFA as in2 to ProdFA as out1
{
   out1.Event(x) :- in1.Event(x); in2.Event(x).
   out1.State(s, t) :- in1.State(s), in2.State(t).
   out1.Current(State(s.name, t.name)) :-
     in1.Current(s), in2.Current(t).

   out1.Transition(State(s, t), Event(a), State(sp, t)) :-
     in1.Transition(State(s), Event(a), State(sp)), in2.State(t).
   out1.Transition(State(s, t), Event(a), State(s, tp)) :-
     in2.Transition(State(t), Event(a), State(tp)), in1.State(s).

   NFAOut := out1.Transition(s, e, sp),
     out1.Transition(s, e, spp), sp != spp.
   notDFAPreserving := in1.DFA.conforms & in2.DFA.conforms &
     out1.ProdFA.conforms & NFAOut.
}
```

Proving Properties

For example, given this partial model:

```
partial model P of DFA
{
   Event(_) Event(_)
   Event(_) Event(_)
   State(_) State(_)
   State(_) State(_)
   Transition(_,_,_) Transition(_,_,_)
   Transition(_,_,_) Transition(_,_,_)
   Current(_)
}
```

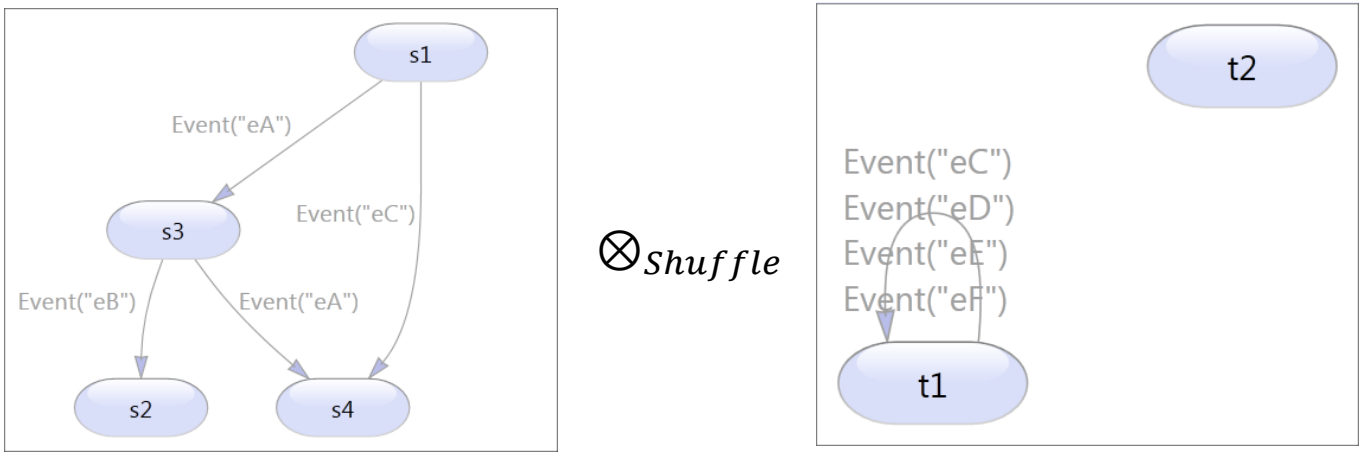
Issue the command:

`solve ShuffleProd P P notDFAPreserving`

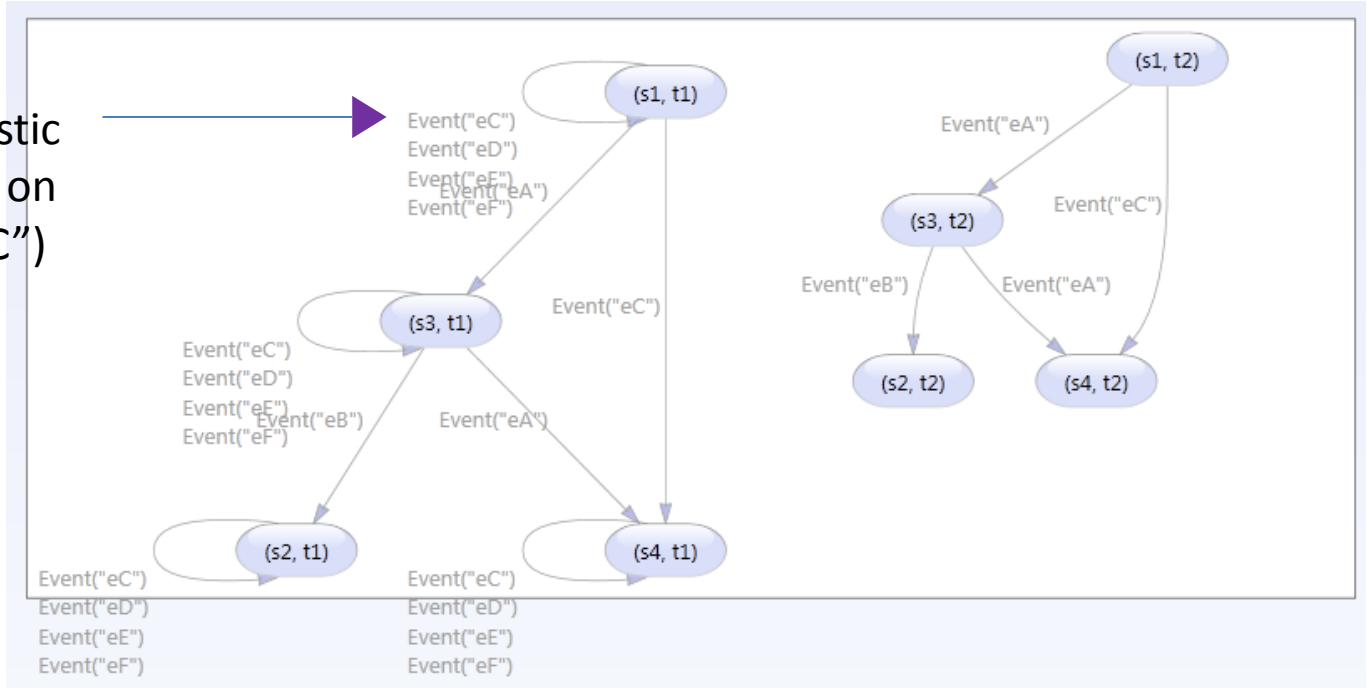
Due to renaming, each P is its own independent copy with different degrees of freedom

Proving Properties

Here is one synthesized counter-example:



Non-deterministic transition on Event("eC")



Modeling Behavior

Transformations are also useful for modeling state evolution.

```
transform Fire<e: in1.Event> from DFA as in1 to DFA as out1
{
   out1.Event(x) :- in1.Event(x).
   out1.State(x) :- in1.State(x).
   out1.Transition(x, y, z) :- in1.Transition(x, y, z).
   out1.Current(sp) :-
    in1.Current(s), in1.Transition(s, e, sp).
   out1.Current(s) :-
    in1.Current(s), fail in1.Transition(s, e, _).
}
```

Bounded symbolic model checking on LTL formulas can be rephrased as model finding over compositions of transformations.

Questions?

http://research.microsoft.com/formula